Operating Systems (2INC0)

Process (02)
Active program execution

Dr. Geoffrey Nelissen
Courtesy of Dr. Tanir Ozcelebi

Interconnected
Resource-aware
Intelligent Systems

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Process concept

- A program is a passive entity
- A process is *a* ***program in execution***
  - Several processes can run the same program
  - User processes run in user mode

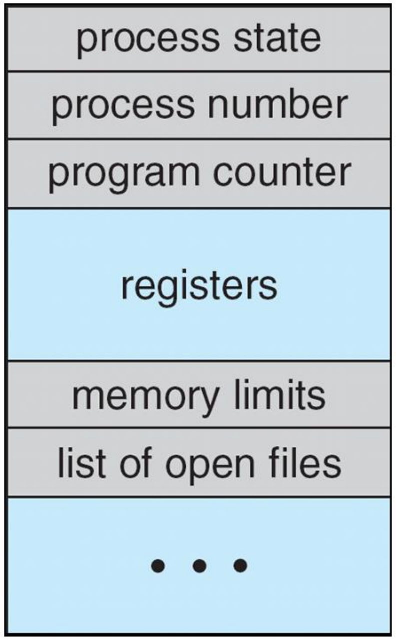
Process Control Block (PCB) – inside kernel

A process is identified by an ID (number) and a pointer to a PCB in kernel space

PCB contains:

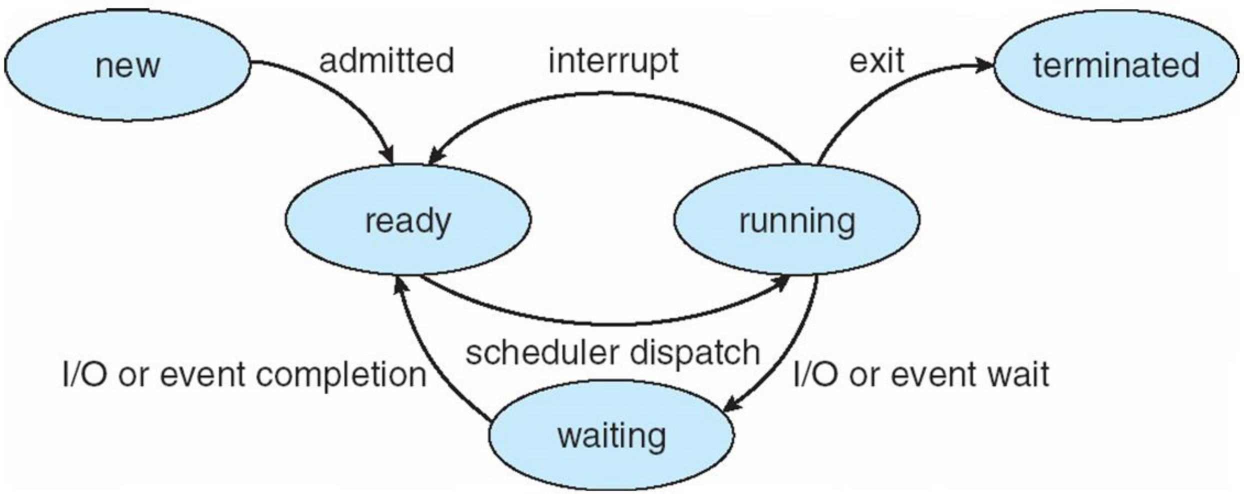
- Process state (see next slide)
- Process number (ID)
- Program counter (needed for context switching)
- CPU registers (needed for context switching)
- Memory management info (values of base, limit registers)
- I/O status information (list of open files, I/O devices)
- Scheduling information (priority, scheduling parameters)
- Accounting information (e.g. CPU time used)
- and more…

PCB



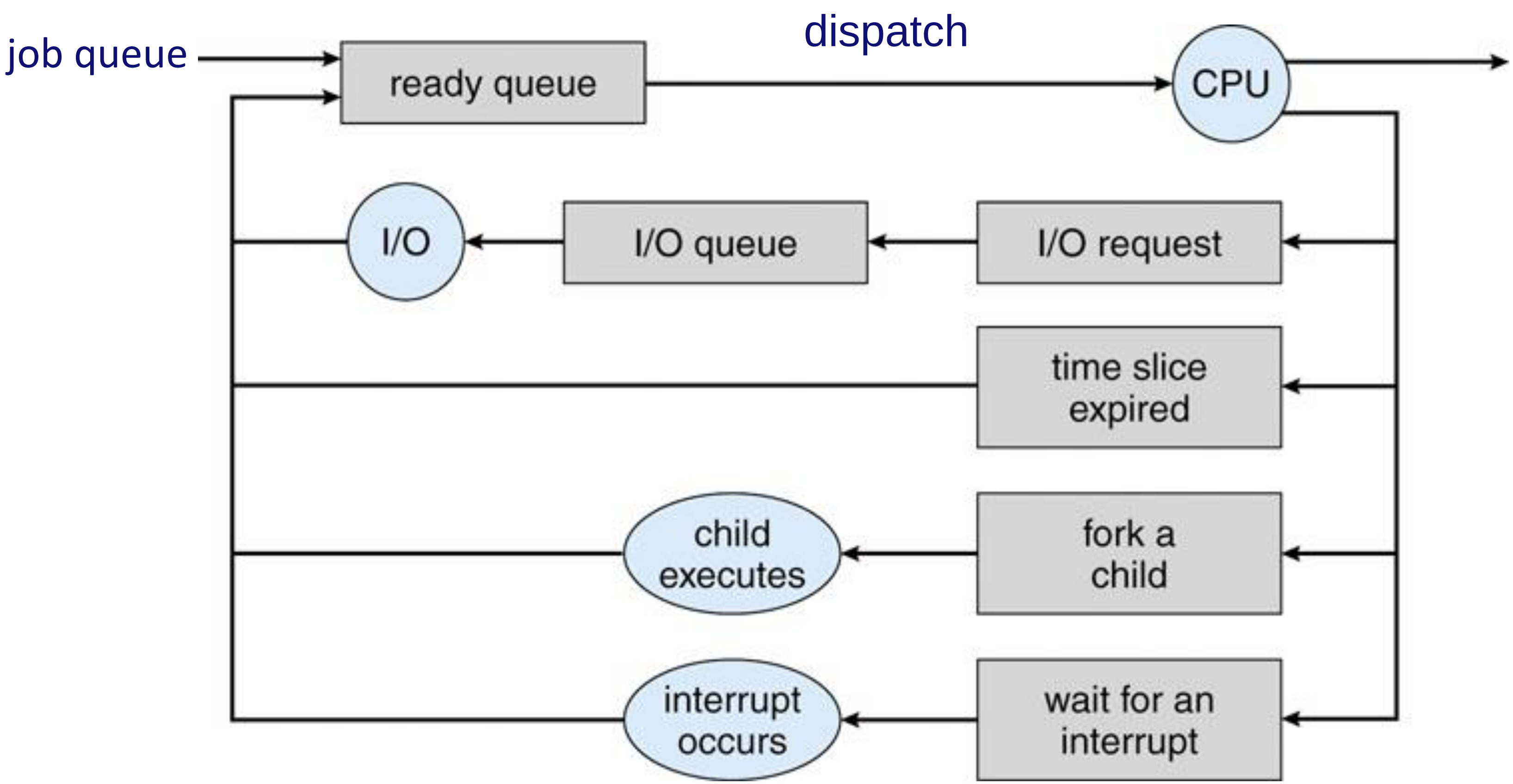
Process states

- new: process is being created
- ready: process is waiting to be assigned to a CPU (before scheduler dispatch)
- running: instructions are being executed
- waiting: process is waiting for some event to occur
- terminated: process has finished execution

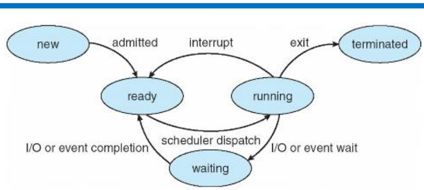


Only one process can run on a CPU at any time instant!
→ must schedule

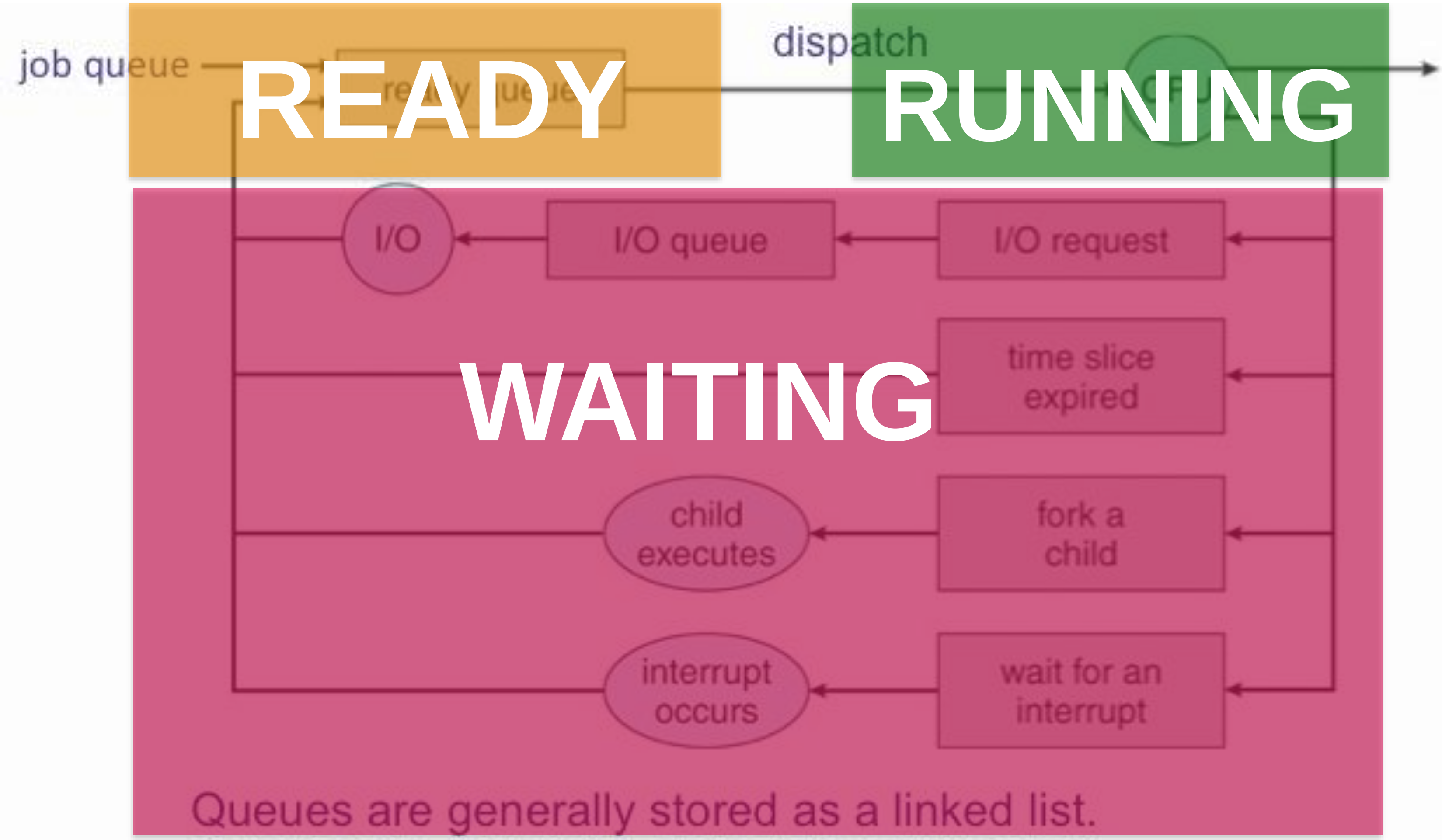
Process scheduling
Ready queue and device (I/O) queues

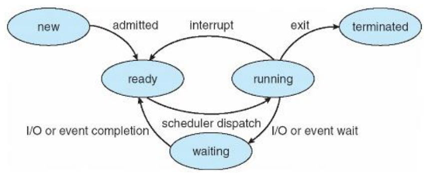


Queues are generally stored as linked lists.



Process scheduling
Ready queue and device (I/O) queues

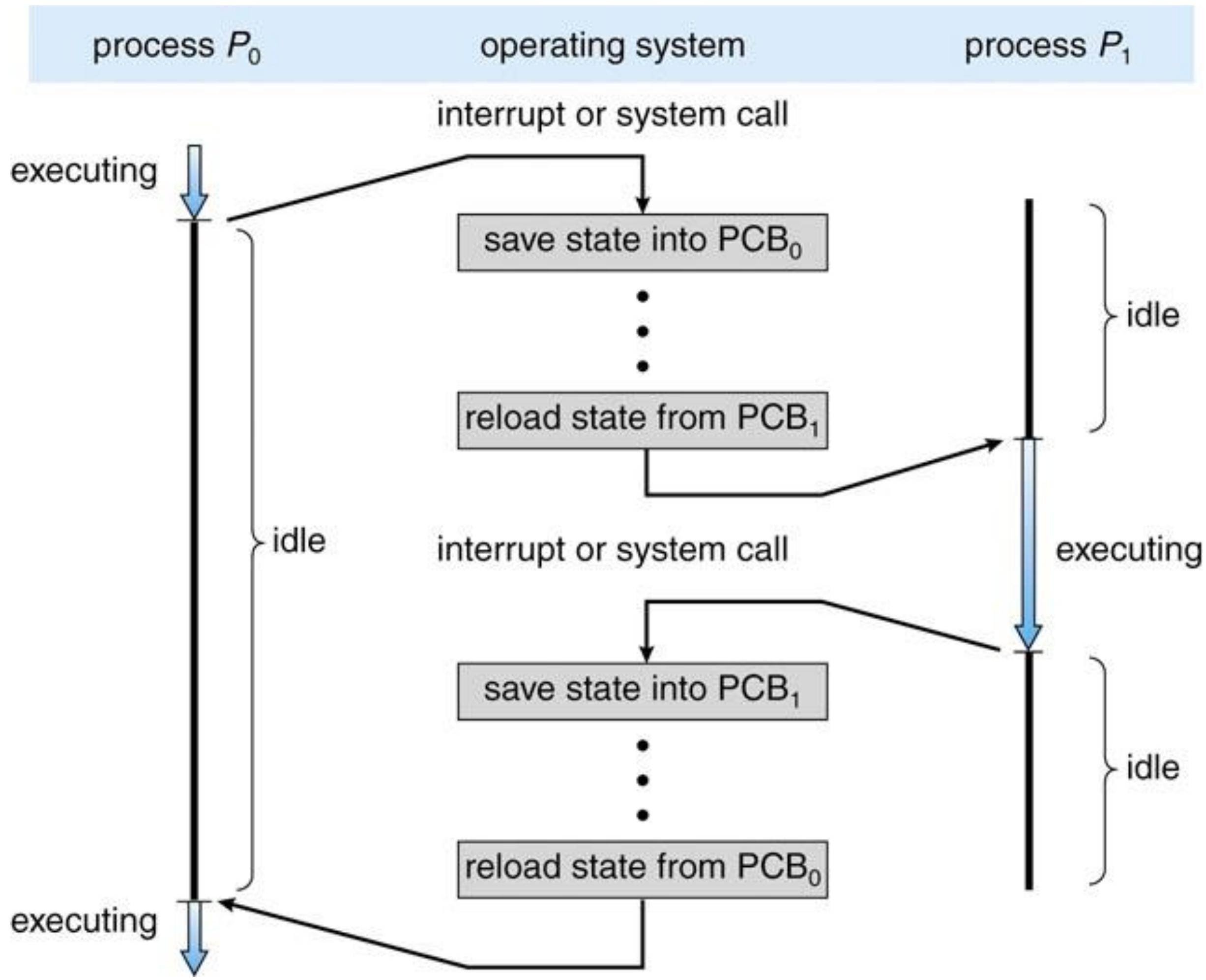




Context switch

Context switch:
Saving the state of a process whose execution is to be suspended, and reloading this state when the process is to be resumed.

Overhead:
(typically) takes a few usecs. → depends on HW



Operating Systems (2INC0)

Process (02)
Process creation/termination

Dr. Geoffrey Nelissen

Courtesy of Dr. Tanir Ozcelebi

Interconnected
Resource-aware
Intelligent Systems

IRiS

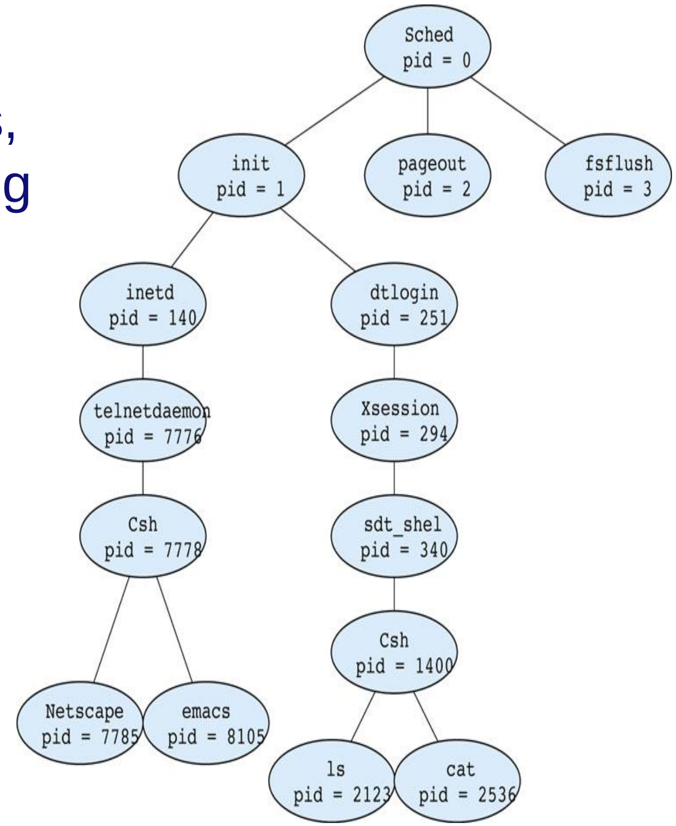
TU/e Technische Universiteit Eindhoven University of Technology

Process Creation

- Starting point is an already running process

- **Parent** process **creates children** processes, which, in turn create other processes, forming a **tree of processes**
  - new processes given process identifiers (pid)

- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources



Process Creation (cont'd)

- Execution options
  - The parent and children **execute concurrently**
  - The parent **waits** until children terminate
- Address space options
  - The child is a **copy** of the parent (e.g., default in Linux)
  - The child has a **new program** loaded into its address space (e.g., Windows)

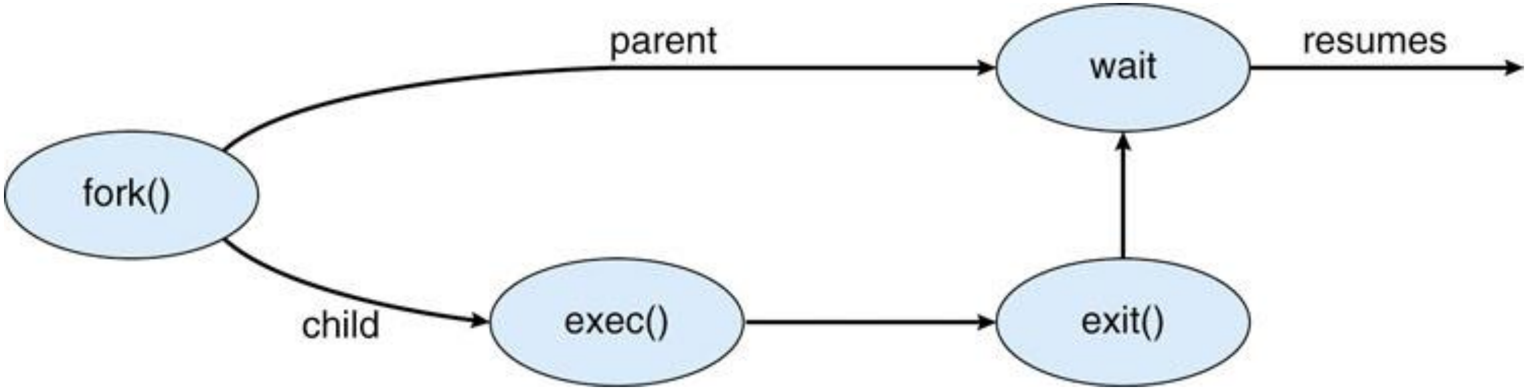
Example: Using the Portable Operating System Interface (POSIX) API

- **IEEE standard on the API** (it's not an OS!)
  - Goal: reduce **portability** effort for applications
  - Many operating systems use the POSIX API (or a subset).

- Implementations may vary …
- … but a system supporting POSIX must provide
  - a host language and compiler (often: C)
  - programming interface definition files (e.g., C-header files)
  - programming interface implementation binary or code (e.g., C-libraries)
  - a run-time system (a platform: OS or the like)

POSIX processes (1003.1)

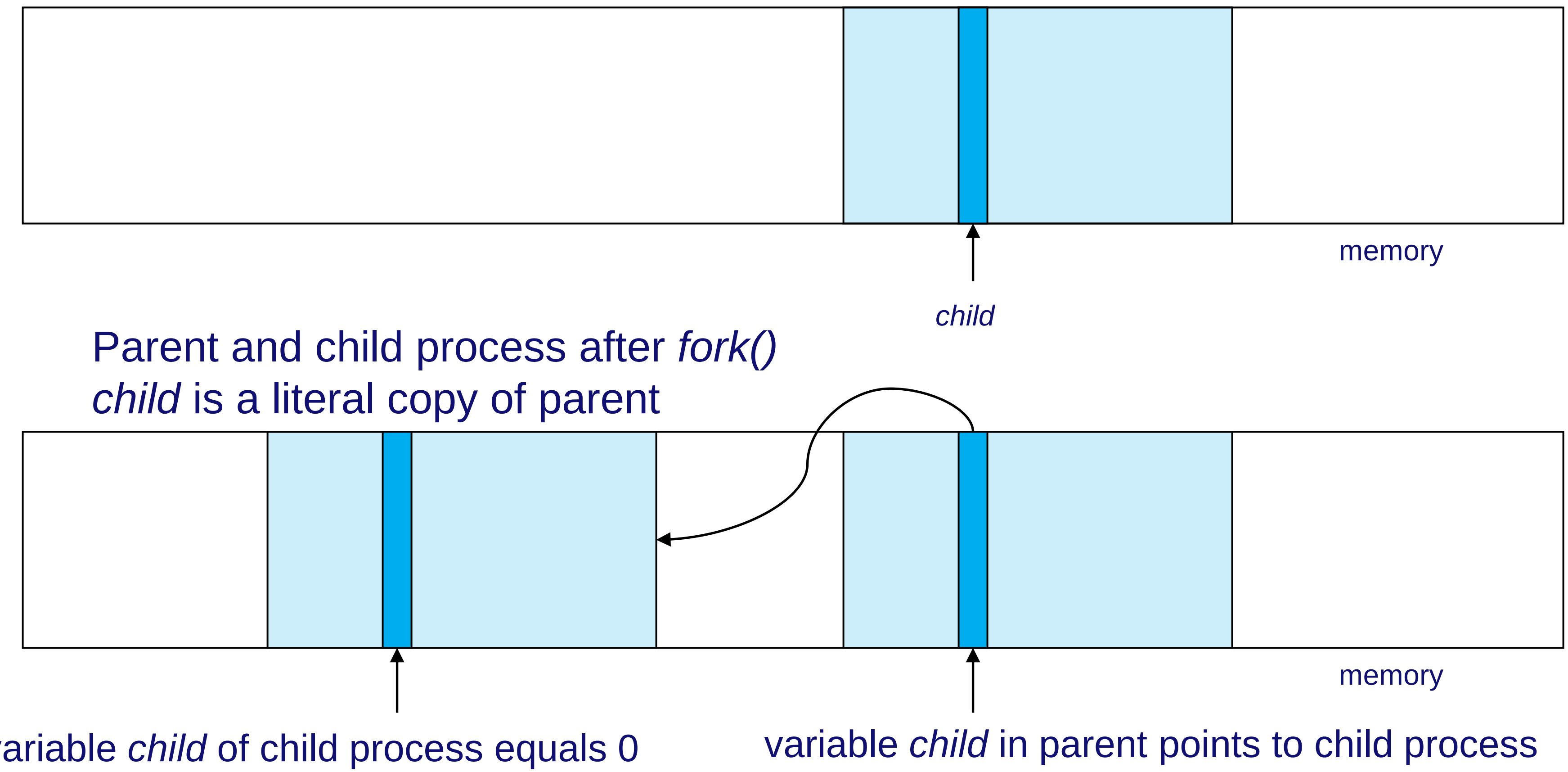
- **POSIX 1003.1 API: *fork() – exec() – exit() – wait()***



- *fork()* function **creates** new process (duplicate address space of the parent in another location)
- *exec()* function used after a fork to write over the process' memory space with a **new program**
- *wait()* function may be issued by the parent to **wait** until the execution of the **child is finished**.

Before and after a *fork()*

Parent process about to perform *child = fork()*



Process Termination

- Process may ask the operating system to delete itself (**exit**)…
  - return status value from child to parent (via wait)
  - the process' resources are taken away by OS

- … or parent may terminate execution of children processes (**abort**)
  - The child has **exceeded allocated resources**
  - The task assigned to child is **no longer required**
  - If the **parent is exiting**
    - Some operating systems do not allow the child to continue if its parent terminates, in which case all children are terminated - **cascading termination**
    - Some operating systems **attach orphans to a 'grandfather' process** (e.g. *init*)

Termination of children: fragment

- Use *exit(status)* to terminate
- *In some OSs, the parent* ***must wait*** *for children* ***to free children's resources***
- Functions *wait(), waitpid()*
  - *wait()* blocks until any child exit
  - *waitpid()* blocks until a specific child changes its state
- Alternative (not shown here): use asynchronous notification signals

```
parent:
    …
    pid_t child, terminated; int status;
    …
    /* blocking wait */
    while (child != wait (&status)) /* nothing */;

   /* or polling wait */
    terminated = (pid_t) 0;
    while (terminated != child) {
       terminated = waitpid (child, &status,
    WNOHANG);
       /* other useful activities */
    }
    /* both cases: status == 23 */
```

```
child:
    ..... exit (23);
```

Operating Systems (2INC0)

Process (02)
Interprocess communication

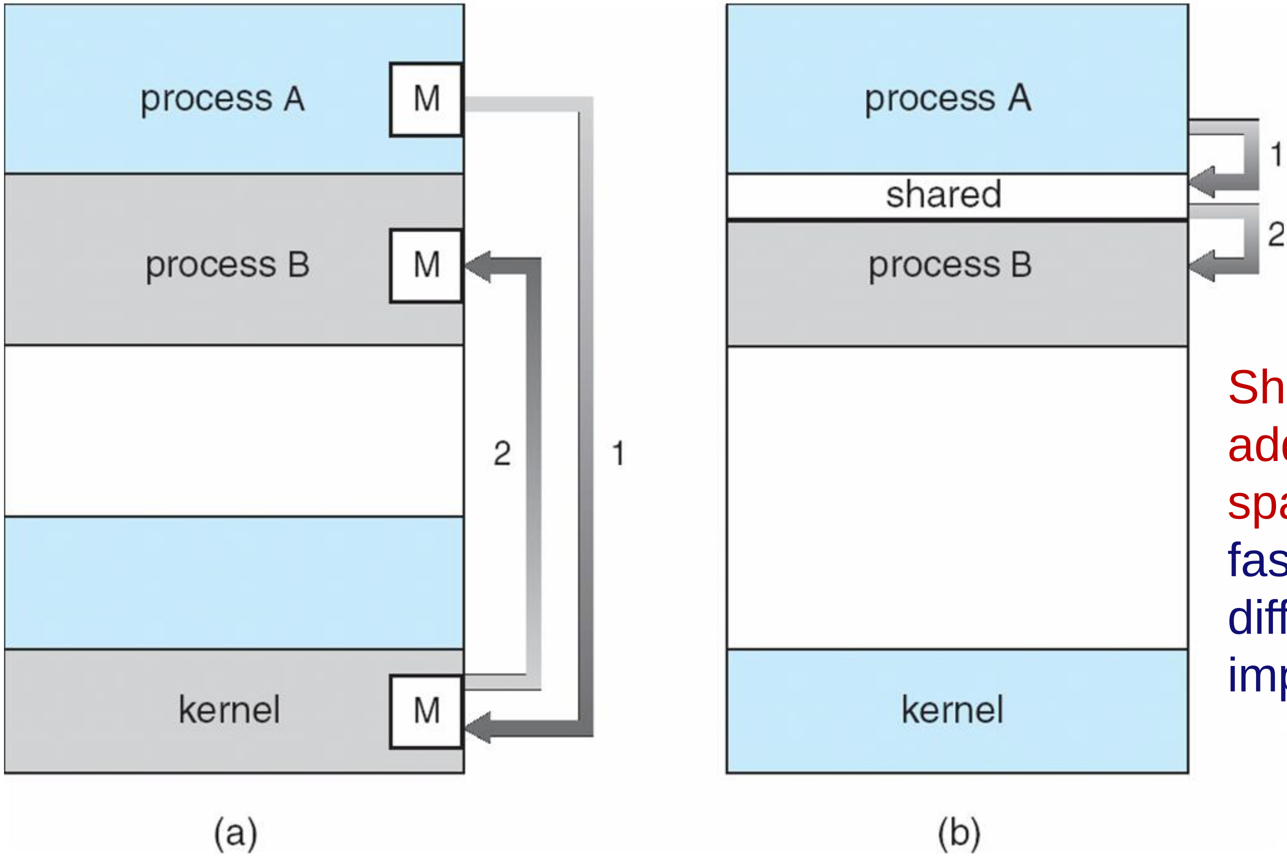
Dr. Geoffrey Nelissen
Courtesy of Dr. Tanir Ozcelebi

IRiS
Interconnected
Resource-aware
Intelligent Systems

TU/e Technische Universiteit Eindhoven University of Technology

Cooperating processes need interprocess communication (IPC)

No shared address space: useful for exchanging small amounts of data



Message passing

Shared memory

Shared address space: faster but difficult to implement

Message passing between processes

- The **easiest** solution especially for **distributed systems**

- The **producer(s) send(s)** messages through a pipe or in a mailbox
  - Can be blocking or non-blocking
- The **consumer(s) read(s)** messages
  - Can be blocking or non-blocking
- If **both send and receive are blocking** then we call it a *rendez-vous*
  - communicating processes synchronize on the transmission

- A temporary queue (**buffer**) can be associated to the communication
  - Allows for **asynchronous executions** and/or applications with **multiple producers/consumers**
  - The producer may or may not be blocked when the buffer is full
    - If non-blocking, the consumers see the **most recent data** and may **miss old ones**(e.g., control systems)

Shared memory between processes

- A memory segment is reserved in the kernel memory
- It is **added to the space addressable** in user mode
- *No kernel involvement is required upon subsequent reading and writing*